

AD-A191 435

NAVAL POSTGRADUATE SCHOOL MONTEREY CA  
INTRODUCTION TO RELATIONAL PROGRAMMING.(U)

F/6 12/1

JUN 81 B J MACLENNAN

UNCLASSIFIED

NP552-81-008

NL

1 of 1  
AD-A191 435

END

DATE

FILED

8-B1

DTIC

AD A101435

14 NPS52-81-008

LEVEL

2

# NAVAL POSTGRADUATE SCHOOL

Monterey, California



6	INTRODUCTION TO RELATIONAL PROGRAMMING
10	Bruce J. MacLennan
11	June 1981
12	32
16	RR00001
17	RR0000110

Approved for public release; distribution unlimited

Prepared for:

Naval Postgraduate School  
Monterey, Ca. 93940

DNC FILE COPY

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----

81 7 15 025

NAVAL POSTGRADUATE SCHOOL  
Monterey, California


Rear Admiral J. J. Ekelund  
Superintendent

D. A. Schradly  
Acting Provost

The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

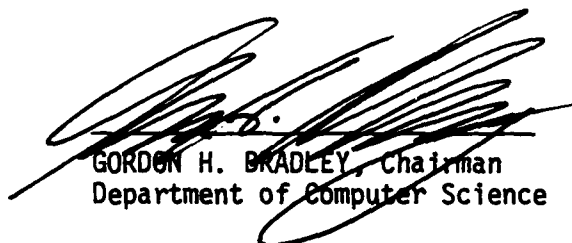
This report was prepared by:



BRUCE J. MacLENNAN  
Assistant Professor of  
Computer Science

Reviewed by:

Released by:



GORDON H. BRADLEY, Chairman  
Department of Computer Science



WILLIAM M. TOLLES  
Dean of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-008	2. GOVT ACCESSION NO. AD-A101435	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Introduction to Relational Programming		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N, RR000-01-10 N0001481WR10034
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE June 1981
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Revised version to be presented at the ACM Conference on Functional Programming Languages and Computer Architecture, October 18-22, 1981.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Relational Programming, Functional Programming, Relational Algebra, Relations, Relational Calculus, Applicative Languages, Combinators, Very-High-Level Languages.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A new method of programming, called <u>relational programming</u> , is introduced. This is a style of programming in which entire relations are manipulated rather than individual data. This is analogous to <u>functional programming</u> , wherein entire functions are the value manipulated by the operators. Because of its ability to manipulate complex data structures other than lists, relational programming seems to have distinct advantages over other very high		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE  
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Level languages. This paper introduces the basic concepts of relational programming and a preliminary notation for expressing them; it does not define a programming language, per se.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

B. J. MacLennan. 81/02/24.

Naval Postgraduate School  
Monterey, CA 93940

## 1. Introduction

In this paper we discuss relational programming, i.e. a style of programming in which entire relations are manipulated rather than individual data. This is analogous to functional programming, wherein entire functions are the values manipulated by the operators. We will see that relational programming subsumes functional programming because every function is also a relation. It is appropriate at this point to discuss why we have chosen to investigate relational programming.

As we have noted, relational programming subsumes functional programming; hence, anything that can be done with functional programming can be done with relational programming. Furthermore, relational programming has many of the advantages of functional programming: for instance, the ability to derive and manipulate programs by algebraic manipulation. A well developed algebra of relations dates back to Boole's original work and has been extensively studied since then. Although relations are more

\* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

general than functions, their laws are often simpler. For instance,  $(fg)^{-1} = g^{-1}f^{-1}$  is true for all relations, but true only for functions that are one-to-one. Also, relational programming more directly supports non-linear data structures, such as trees and graphs, than does functional programming. In relational programming the basic data values are themselves relations, whereas in functional programming there is a separate class of objects (lists) used for data structures. One final reason for investigating relational programming is that it provides a possible paradigm for utilizing associative and active memories. As a teaser for what is to come, we present the following example of a relational program. We will take a text  $T$ , represented as an array of words (i.e.,  $T:i$  is the  $i$ -th word), and generate a frequency table  $F$  so that  $F:w$  is the number of occurrences of word  $w$  in  $T$ . Now we will see (section 3) that  $\hat{T}:w$  is the set of all indices of the word  $w$ . If we let  $\# : C$  be the cardinality of a class, then the number of indices (occurrences) of  $w$  is just  $\# : (\hat{T}:w)$ . Therefore we can write  $F = \# \hat{T}$  (section 6). For a second example, we will see in section 13 that a program to update a payroll file  $\Phi$  can be written:

$$\Phi' = Md_{\Phi}^{\Phi} \quad \text{where } u = (,H)(+) \frac{(:H)\Phi}{u}$$

## 2. Classes and Relations

As is usual we will use  $xRy$  to mean that  $x$  bears the rela-

tion  $R$  to  $y$ . Similarly, we will write  $x_C$  or  $x \in C$  to mean  $x$  is in the class  $C$  (i.e.  $x$  has the property  $C$ ). Our theory of relations will be typeless, like that described in [6]; this seems more appropriate to programming than systems such as Russell's "ramified type theory." In most other respects our notation follows that of Carnap [1] and Whitehead and Russell [8]. There is no special significance to the case of variables, although we will often distinguish relations (and classes) from the things they relate by putting them in upper case.

We often need to talk of the individuals that can occur on the right or left of a relation. We say that  $x$  is a left-member of  $R$  whenever there is a  $y$  such that  $xRy$ .

$$x \text{ Lm } R \iff \exists y(xRy)$$

Right-member and member ( $Rm$  and  $Mm$ ) are defined analogously.

### 3. Functions

We define functions as special classes of relations: A relation  $F$  is a function if and only if it is left-univalent:

$$F \text{ lun} \iff \forall xyz[ yFx \wedge zFx \Rightarrow y=z ]$$

If  $F$  is left-univalent then we can write  $F:x$  for the unique  $y$  such that  $yFx$  (if such a  $y$  exists). This differs from the usual convention, in which  $y=F:x$  means  $xFy$ , but agrees with [8] and



works better with the rest of the notation. Right-univalent and bi-univalent relations (run and bun) are defined analogously.

The fact that  $F:x$  may be meaningless makes it convenient to use several other relations derived from  $F$ . One of these is the plural description. If  $F$  is any relation and  $C$  is a class then  $F!:C$  is the set of all  $y$  such that  $yFx$  for some  $x$  in  $C$ , i.e.,

$$y \in F!:C \iff \exists x(yFx \wedge xC)$$

Notice that the operation  $F!:C$  is defined for all relations  $F$  and classes  $C$ , regardless of whether  $F \in \text{un}$  or the members of  $C$  are right members of  $F$ .

Related ideas are the image and converse image of an individual. If  $R$  is a relation, then  $c\vec{R}x$  means that  $c$  is the class of individuals related to  $x$ . This class is called the image of  $x$ , and is defined:

$$y \in \vec{R}:x \iff yRx$$

The analogous idea is that of the converse image of  $y$ :

$$x \in \overleftarrow{R}:y \iff yRx$$

Like the plural description,  $\vec{R}$  and  $\overleftarrow{R}$  are defined for all  $R$  and all arguments. It is generally safer to use  $\vec{f}$  than  $f$  since  $f:x$  may be undefined whereas  $f:x$  is always defined.

#### 4. Combining Relations

We will next investigate ways of combining relations and classes. The simplest methods are just abstractions of the logical connectives used between propositions: intersection, union, negation and difference ( $\wedge$ ,  $\vee$ ,  $-$ ). For instance  $R \vee S$  is defined so that:

$$x(R \vee S)y \iff xRy \vee xSy$$

As an example of the use of these operations, the definition of  $Mm$  can be written:

$$Mm = Lm \vee Rm$$

The logical connectives satisfy the usual properties of a Boolean algebra (e.g., DeMorgan's theorem).

We will also define the converse of a relation. The relation  $R^{-1}$  is called the converse of  $R$ , i.e.  $xR^{-1}y \iff yRx$ . If  $f$  is a function then  $f^{-1}$  is the inverse of  $f$ .

#### 5. Limiting and Restriction

It is often useful to limit the left or right domain of a relation. Consider the relation  $x \sin^{-1} y$ , which means that  $x$  is an arcsine of  $y$ . We cannot write  $x = \sin^{-1}y$  because  $\sin^{-1}$  is not left univalent (i.e. it is not a function). We can solve this problem by defining a function  $\text{Sin}$  whose arguments are

restricted to be in the range  $-\pi/2$  to  $\pi/2$ . Let  $S$  be the class of reals in this range:

$$S = (\vec{\geq}:-\pi/2) \wedge (\vec{\leq}:\pi/2)$$

then we will write  $\sin\upharpoonright S$  for the sine function with its arguments restricted to  $S$ , which is exactly the  $\text{Sin}$  function we sought:

$$\text{Sin} = \sin\upharpoonright[(\vec{\geq}:-\pi/2) \wedge (\vec{\leq}:\pi/2)]$$

This function is bi-univalent, so it is invertible. It is now perfectly meaningful to write  $\text{Sin}^{-1}:y$  (if  $y \text{ Lm } \sin$ ). The right-restriction operation is defined:

$$x(R\upharpoonright S)y \leftrightarrow xRy \wedge yS$$

The left-restriction,  $S\upharpoonleft R$ , is defined analogously. These notations can be combined to restrict both domains:  $S\upharpoonleft R\upharpoonright T$ . The combination  $S\upharpoonleft R\upharpoonright S$  is so common that a special restriction notation is provided for it:

$$r\upharpoonright s = s\upharpoonleft r\upharpoonright s$$

For instance,  $\text{pred}\upharpoonright(\vec{>};0)$ , is the predecessor relation restricted to positive numbers.

## 6. Relative Product and Composition

If  $R$  is the relation "... is a son of ..." and  $S$  is the relation "... is a brother of ...", then the relative product,

$R|S$ , is the relation "... is a son of a brother of ...". More formally,

$$x(R|S)z \leftrightarrow \exists y(xRy \wedge ySz)$$

Where there is little chance of confusion, we will write  $RS$  for  $R|S$ . If  $f$  and  $g$  are functions it is easy to see that  $f|g$  is the composition of these functions. Hence,  $fg:x = f:(g:x)$ .

It is convenient to have a notation for relative products of a relation with itself. For instance, the "grandparent" relation can be written  $\text{parent}|\text{parent}$ , which we abbreviate  $\text{parent}^2$ . In general,

$$R^1 = R$$

$$R^{n+1} = (R^n)R = R(R^n)$$

$$R^{-n} = (R^n)^{-1} = (R^{-1})^n$$

$$R^0 = \exists \vec{x}(\vec{Mm}:R)$$

It is easy to show these properties of the relative product:

$$(rs)t = r(st)$$

$$r(s \vee t) = rs \vee rt$$

$$(r \vee s)t = rs \vee rt$$

$$r(s \wedge t) \subseteq rs \wedge rt$$

$$(r \wedge s)t \subseteq rt \wedge st$$

$$\exists(rs) \leftrightarrow \exists(\vec{Rm}:r \wedge \vec{Lm}:s)$$

$$(r^{-1})^{-1} = r$$

$$(rs)^{-1} = (s^{-1})(r^{-1})$$

$$r^m r^n = r^{m+n} \quad (m, n \geq 0)$$

$$(r^m)^n = r^{mn} \quad (m, n \geq 0, \text{ or } r \in \text{bun})$$

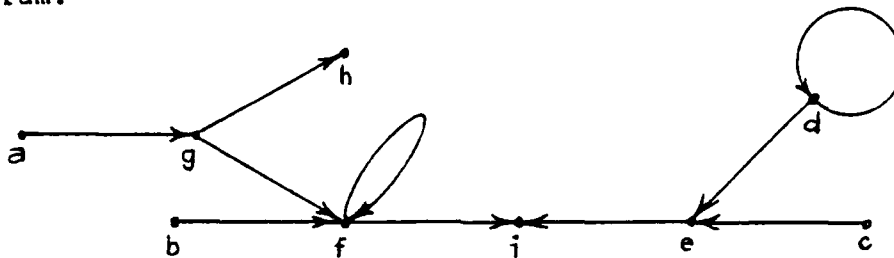
$$r^m r^n \subseteq r^{m+n} \quad (r \in \text{bun})$$

$$rr^{-1} = r^{-1}r = r^0 \quad (r \in \text{bun})$$

## 7. Structures

### 7.1 initial and terminal members

Suppose  $R$  is the relation represented by the following diagram:



Since  $a = R:g$  and  $g = R^{-1}:a$  it can be seen that  $R^{-1}$  follows an arrow and  $R$  goes against an arrow. Now, notice that the left and right members of  $R$  are:

$$\vec{Lm}:R = \{ a, b, c, d, e, f, g \}$$

$$\vec{Rm}:R = \{ g, f, e, d, i, h \}$$

We define the initial members of  $R$  to be those members which are not pointed at by an arrow. Therefore, the initial members of  $R$

are the left members that are not right members.

$$\text{init:R} = \overrightarrow{(\text{Lm}-\text{Rm})}:R = \{a, b, c\}$$

The terminal members of a relation do not point to other members:

$$\text{term:R} = \overrightarrow{(\text{Rm}-\text{Lm})}:R = \{h, i\}$$

When a relation is used to represent a data structure, the above functions become important.

For instance, a sequence is represented by a relation with the structure:

$$S = \begin{array}{ccccccc} a_1 & a_2 & a_3 & \dots & a_{n-1} & a_n \\ \bullet & \rightarrow & \rightarrow & \dots & \rightarrow & \rightarrow \end{array}$$

In this case  $\text{init:S}$  is the unit class containing the first element of the sequence (i.e.,  $a_1$ ) and  $\text{term:S}$  is the unit class containing the last element of the sequence (i.e.,  $a_n$ ). Similarly,  $S \setminus (-\text{init:S})$  is the sequence with its first element deleted. Hence, the following common sequence manipulation functions can be defined:

$$\begin{aligned} \alpha:S &= \theta \text{ init: } S, & \text{"first"} \\ \omega:S &= \theta \text{ term: } S, & \text{"last"} \\ \Omega:S &= S \setminus (-\text{init:S}), & \text{"final"} \\ A:S &= (-\text{term:S}) \setminus S, & \text{"initial"} \end{aligned}$$

where  $\theta$  selects the element of a singleton set ( $\theta = (\overset{\rightarrow}{=^{-1}})$ ). More

operations on sequences are discussed in the next section.

As another example of the use of 'init' and 'term', consider a relation  $T$  representing a tree. Then,  $\theta \text{ init}: T$  is the root of the tree, and  $\text{term}: T$  is set of the leaves of the tree. The result is analogous for forests. Given a forest  $F$  the set of roots is  $\text{init}: F$  and the set of leaves is  $\text{term}: F$ .

## 7.2 higher level operations

The set of nodes directly descended from  $n$  is just  $\overrightarrow{F^{-1}}:n$ . For instance, the set of nodes directly descended from a root is  $F^{-1}!\text{init}:F$ . and the set of nodes that point to leaves is  $F!\text{term}:F$ .

These operations can be used for obtaining the maximum and minimum of sets. Suppose ' $<$ ' is the less-than relation on integers and  $S$  is some set of integers. Then  $<\mathbb{X}S$  is the less than relation restricted to this set, i.e. it is a sorting of the set. Now note that  $\alpha:(<\mathbb{X}S)$  and  $\omega:(<\mathbb{X}S)$  are the minimum and maximum elements of the set:

$$\text{min}:S = \alpha:(<\mathbb{X}S)$$

$$\text{max}:S = \omega:(<\mathbb{X}S)$$

These operations are only defined if  $S$  has two or more elements, since an irreflexive relation cannot relate less than two elements. That is, an irreflexive relation when restricted to a

unit or empty class becomes the empty relation. Notice that we can select the maximum and minimum based on any relation that is a series (i.e., transitive, irreflexive and connected).

The following are simple properties of these operations:

$$\text{init}:r = \text{term}:(r^{-1})$$

$$\text{term}:r = \text{init}:(r^{-1})$$

$$\text{init}:(r \times s) = \text{term}:(r^{-1} \times s)$$

## 8. Sequences

### 8.1 pairs

In this section we will continue the discussion of sequences begun in the last section. We saw that it was easy to define the selector functions on sequences ( $\alpha$ ,  $\omega$ ,  $A$ ,  $\Omega$ ). This provides us with functions for taking sequences apart. We will define the ordinal couple or pair, which puts them together. If  $x$  and  $y$  are two objects, then ' $x,y$ ' is the relation that relates  $x$  and  $y$  but no other objects.

$$(x,y) = \begin{array}{c} \bullet \longrightarrow \bullet \\ x \qquad y \end{array}$$

Observe that

$$\alpha:(x,y) = x$$

$$\omega:(x,y) = y$$



It will occasionally be convenient to write ordinal couples in a vertical format:

$$\begin{pmatrix} x \\ y \end{pmatrix} = (x, y)$$

The class of all the ordinal couples (or pairs) that can be made from the classes S and T is SXT:

$$p \in (S \times T) \iff \exists xy [x \in S \wedge y \in T \wedge p = (x, y)]$$

## 8.2 catenation and consing

If s and t are sequences then we can define an operation 's^t', which is the catenation of s and t. To form this catenation we must hook the last element of s to the first element of t:

$$s^t = s \vee (\omega:s, \omega:t) \vee t$$

The catenation operation is only defined for sequences, which are required to have at least two elements (since an irreflexive relation with less than two elements is the empty relation).

How do we add a single element to the left or right of a sequence? The "cons left" and "cons right" operations are easy to define:

$$x \text{ cl } s = (x, \omega:s) \vee s$$

$$s \text{ cr } y = s \vee (\omega:s, y)$$

It is easy to show that if s is a sequence, then:

$$\alpha:(x \text{ cl } s) = x$$

$$\Omega:(x \text{ cl } s) = s$$

$$u:(s \text{ cr } y) = y$$

$$A:(s \text{ cr } y) = s$$

Also, if  $s$  is a sequence, then  $s \vee (u:s, \alpha:s)$  is a ring formed by joining the last element of  $s$  to the first element.

If  $s$  is a sequence, then  $s^{-1}$  is the reverse of  $s$ . Hence,

$$\alpha:s = u:s^{-1}$$

$$u:s = \alpha:s^{-1}$$

$$A:s^{-1} = (\Omega:s)^{-1}$$

$$\Omega:s^{-1} = (A:s)^{-1}$$

$$(s^{\wedge}t)^{-1} = t^{-1} \wedge s^{-1}$$

$$(x \text{ cl } s)^{-1} = s^{-1} \text{ cr } x$$

$$(s \text{ cr } x)^{-1} = x \text{ cl } s^{-1}$$

$$(x,y)^{-1} = (y,x)$$

$$\begin{pmatrix} x \\ y \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} x \\ z \end{pmatrix}$$

## 9. Binary Operations

In this section we will discuss our approach to binary operations — that is, to functions with two arguments and one result. We have already seen how unary functions are connected

to relations. For instance, we can write the fact that  $y$  is the sine of  $x$  by either: ' $y \sin x$ ' or  $y = \sin:x$ . Since we only deal with binary relations, we will have to have a new convention for handling binary functions. This convention is: we will combine the two arguments of an operation into a pair. For instance, we can define a relation 'sum' such that

$$x \text{ sum } (y,z)$$

if and only if  $x$  is the sum of  $y$  and  $z$ . We can use our colon convention as usual, e.g.,

$$x = \text{sum}:(y,z) \quad \leftrightarrow \quad x \text{ sum } (y,z)$$

Now, it would be inconvenient to have to invent names, such as 'sum', for each operation, such as '+'. Hence, we will adopt a systematic convention for making such names: either placing the conventional infix symbol for the operation in bold face or in parentheses. For instance,

$$\begin{aligned} x \underline{+}(y,z) &\leftrightarrow x = \underline{+}:(y,z) \leftrightarrow x = y+z \\ x (+) (y,z) &\leftrightarrow x = (+):(y,z) \leftrightarrow x = y+z \end{aligned}$$

This notation will permit us to manipulate in a more regular fashion the usual arithmetic operations (+, -, \*, /) as well as the relational operations ( $\wedge$ ,  $\vee$ , etc). For instance, if  $S$  is a class of classes, then

$$(\wedge)!:SXS$$

is the class of all pairwise intersections of members of S.

It is often convenient to be able to generate simple relations from a binary operation. Following Russell and Whitehead, let  $\mathbb{W}$  represent any binary operation. We define:

$$x(\mathbb{W}z)y \leftrightarrow x = y\mathbb{W}z$$

$$x(y\mathbb{W})z \leftrightarrow x = y\mathbb{W}z$$

Hence,

$$x(-1)y \leftrightarrow x = y-1$$

therefore  $(-1)$  is the predecessor relation. These can be used as functions:

$$(-1):x = x-1$$

$$(+1):x = x+1$$

This convention makes it very easy to form more complex functions. For instance, if we want  $f:x = \sin:(1/x)$  then we can define  $f = \sin(1/)$  To see that this works:

$$f:x = [\sin(1/)]:x = \sin:[(1/):x] = \sin:[1/x]$$

## 10. Combinators

In this section we will discuss several powerful operations for manipulating relations. These are called combinators because of their similarity to the combinators of Curry and Feys [4].

First observe the action of the  $(x,)$  and  $(,y)$  functions:

$$(x,):y = (x,y)$$

$$(,y):x = (x,y)$$

Now note that

$$f(x,):y = f:[(x,):y] = f:(x,y)$$

In general, if  $f$  is a binary function, then  $f(x,)$  and  $f(,y)$  are the "partially instantiated" unary functions. This is the effect of Curry and Feys "B" combinator [4], the elementary compositor.

If  $' : R = R^{-1}$  then, since  $S^{-1}$  is the reverse of a sequence,  $'$  is the reverse form of an operation. For instance,  $(-)'$  is the reverse subtract operation:

$$\begin{aligned} (-)':(x,y) &= (-):(':(x,y)) \\ &= (-):(y,x) \\ &= y-x \end{aligned}$$

Thus  $(-)'$  can be read "subtract from" and  $(/)'$  can be read "divide into". This is Curry and Feys "C" combinator, or elementary permutator.

The next combinator we will discuss is the paralleling of relations,  $\frac{R}{S}$ , which is defined:

$$\left( \frac{R}{S} \right) (x,y) \leftrightarrow uRx \wedge vSy$$

So, if  $f$  and  $g$  are functions,

$$\left(\frac{f}{g}\right): \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f:x \\ g:y \end{pmatrix}$$

Hence,  $\frac{f}{g}$  is the element-wise combination of  $f$  and  $g$ .

Another combinator is the elementary duplicator,  $W$ , defined so that

$$(W:f): x = f:(x,x)$$

If we define  $\Delta:x = (x,x)$  then it is easy to see that  $W:f$  is just  $f\Delta$ . For instance,  $(x)\Delta$  is the squaring function:

$$(x)\Delta:n = (x):(\Delta:n) = (x):(n,n) = nxn = n^2$$

It should be clear that Backus'  $[f,g]$  combining form is just our  $\frac{f}{g}\Delta$ , since

$$\frac{f}{g}\Delta : x = \left(\frac{f}{g}\right): \begin{pmatrix} x \\ x \end{pmatrix} = \begin{pmatrix} f:x \\ g:x \end{pmatrix}$$

Since this combination is so common we will adopt a special notation for it:

$$\frac{f}{g} \mid = \frac{f}{g} \Delta$$

$$\text{Hence, } \frac{f}{g} \mid : x = \begin{pmatrix} f:x \\ g:x \end{pmatrix}$$

Some of the properties satisfied by these combinators are:

$$\begin{aligned}
 \frac{R}{S} \frac{T}{U} &= \frac{RT}{SU} \\
 \left( \frac{R}{S} \right)^n &= \frac{R^n}{S^n} \\
 \frac{R}{S} \Big| T &= \frac{RT}{ST} \\
 \frac{R}{S} \frac{T}{U} \Big| &= \frac{RT}{SU} \\
 \Delta^R &= \frac{R}{R} \\
 \left( \frac{R}{S} \right)' &= \frac{S}{R} = \left( \frac{R}{S} \right) \\
 \left( \frac{R}{S} \right)' \Big| &= \frac{S}{R} \\
 \propto \frac{R}{S} &= R \downarrow (\overrightarrow{Rm:g}) \\
 \omega \frac{R}{S} &= S \downarrow (\overrightarrow{Rm:f}) \\
 \frac{R}{S} &= \frac{Rcc}{SUU} \\
 cl &= \frac{cc}{R}^{-1} \\
 cr &= \frac{A}{UU}^{-1}
 \end{aligned}$$

As an example of these combinators it is easy to show that

$$f = (+) \frac{(x)\Delta}{2x}$$

is the function  $f:t = t^2+2t$ .

Our last combinator is the meta-application operator,  $::$ , which corresponds to Curry and Feys' S combinator:

$$(f::g):x = (f:x):(g:x)$$

For instance,  $[(!)]::init$  is the operation that gives the set of descendents of roots of a forest,  $F$ , since  $((!)]::init):F = (F^{-1}!):(init:F)$ .

## 11. Arrays

An array is just a function from a contiguous subset of the integers to some set of values. If  $A$  is an array and  $i \in \text{Rm } A$  then  $A[i]$  is the  $i$ -th element of  $A$ . Similarly, if  $I \subseteq \text{Rm } A$  is a set of index values then  $A[I]$  is the corresponding set of array values and  $A[I]$  is the subarray of  $A$  selected by those indices.

It is easy to define multi-dimensional arrays: they are just arrays whose elements are selected by sequences of integers, e.g.  $M[i,j]$ . If  $M$  is a two-dimensional array, then  $M[i,]$  is the  $i$ -th row of  $M$  and  $M[,j]$  is the  $j$ -th column of  $M$ . Also, if  $I$  is a set of row indices and  $J$  is a set of column indices then  $M[I,J]$  is the submatrix of  $M$  selected by these sets. It is easy to see that  $M'$  is the transpose of  $M$ , since

$$M'[i,j] = M[:,(i,j)] = M[j,i]$$

More generally, if  $P$  is a permutation function (i.e. a bijection from an index set into itself) then  $AP$  is the result of permuting  $A$  by  $P$ .

Suppose  $x$  is an element of the array  $A$  (i.e., for some  $i$ ,  $x=A[i]$ ). Then  $\bar{A}:x$  is the set of all indices for which  $x=A[i]$ . Therefore we can find the index of the first occurrence of  $x$  in  $A$  (i.e. APL's iota operator) by  $\min \bar{A}:x$ . In general, if  $P$  is some property (i.e. class), then  $A^{-1}!P$  is the set of indices of all elements of  $A$  that satisfy  $P$ . A sorted reflexive sequence of



these indices is just  $\leq x (A^{-1}! : P)$

It is easy to convert arrays to sequences and vice versa. Suppose all the elements of A are distinct, then  $A^{-1}$  is a function that returns the index of an element of A. We want to define a sequence S such that  $xSy$  if and only if x precedes y in A, i.e. the index of x is one less than the index of y.

$$\begin{aligned} xSy &\leftrightarrow (A^{-1}:x) = (A^{-1}:y) - 1 \\ &\leftrightarrow (A^{-1}:x) (-1) (A^{-1}:y) \\ &\leftrightarrow x[A(-1)A^{-1}]y \end{aligned}$$

Hence,  $S = A(-1)A^{-1}$ .

We will finish our discussion of arrays by investigating the generation of sorted arrays. Let S be a set of integers to be sorted, then  $\leq xS$  is a structure which relates lesser elements to greater elements. Now if x is any element of the set,  $(\overrightarrow{\leq xS}):x$  is the set of all elements less than x. Thus  $[\#(\overrightarrow{\leq xS})]:x$  is the number of elements of S less than or equal to x. This is just the index of x in the sorted array we seek. Hence if A is the sorted array,  $xAi$  if and only if  $i[\#(\overrightarrow{\leq xS})]x$ , so  $A = [\#(\overrightarrow{\leq xS})]^{-1}$ . Of course this can be generalized to any ordering relation.

## 12. Scanning Structures

It is often useful to scan a structure while performing some processing at each node. When the data structure is a sequence

this amounts to APL's reduce operator and Backus' insert operator. We will define a scanning operation that works on a more general class of structures. This operator can be understood intuitively as follows: The state of the scanning process is represented by a set of "read heads" each of which is "positioned over" a node and holds state information accumulated from the nodes it has already visited. A node can be processed when a read head has moved to that node over each edge which leads into the node. When this occurs a processing function is applied to the node (as first parameter) and the union of the state information of each of the read heads (as second parameter). The result of this processing step becomes the state information associated with a new set of read heads which are advanced along each edge leading out from the node. The processing of the structure is completed when all read heads have arrived at terminal nodes (hence this scanning operation is not defined for cyclic structures). Scanning a structure is started by positioning a read head with initial state information over each initial node.

The scanning operation is symbolized by  $f[i]$ , where  $f$  is the processing function and  $i$  is the initial state for the read heads. For instance, if  $V$  is a vector,  $(+)[0:V]$  will scan the elements of  $V$  using  $(+)$  (i.e. APL  $+/V$  or Backus'  $(/+) : V$ ). For a more interesting example, suppose  $T$  is an attributed parse tree,  $E$  is a function that evaluates attributes and  $B$  is the initial set of attribute bindings. Then  $E[B:T]$  propagates the values of

inherited attributes down to the leaves of the tree. Conversely,  $E[B:(T^{-1})]$  propagates the values of synthesized attributes back to the root. Hence, repeated applications of  $E[B]$  and  $(E[B])'$  will evaluate all of the attributes. Of course, this program will work just as well if  $T$  is a forest of parse trees. The  $I$  operator is still undergoing evaluation as it is one of several possible structure-directed scanning operations.

### 13. Examples

In this section we will give several examples of relational programs.

PAYROLL EXAMPLE: Suppose we have a file  $\Phi$  of employee records, where  $r = \Phi:n$  is the record for the employee with the employee number  $n$ . We will suppose that employee records are functions defined so that:

$r:N$  = employee name  
 $r:H$  = hours worked so far this week  
 $r:R$  = pay rate

We are given an update file  $U$  such that  $U:n$  is the number of hours worked by employee  $n$  today. We wish to generate a new payroll file  $\Phi'$ .

SOLUTION: Let  $r = \Phi:n$  and  $r' = \Phi':n$  be the old and new employee records. It is clear that  $r'$  is the same as  $r$  except

for its H field. In order to modify part of a relation, we will use the Md function defined by:

$$Md:(S,R) = R \vee S \uparrow (-\vec{R}m:R)$$

Then, if h' represents the new value of the H field, the new employee record is

$$r' = Md:(r, (h', H))$$

Now, h' is just the cumulative hours worked:

$$h' = (\phi:n):H + U:n$$

By combining these results we have,

$$\phi':n = r' = Md:(\phi:n, (h', H))$$

To find  $\phi'$  we must factor out the employee number n. To do this, note that  $(\phi:n):H = (:H):(\phi:n) = (:H)\phi:n$ . That is,  $(:H)\phi$  is a slice of the payroll file: the hours worked for each employee. Therefore,

$$\begin{aligned} h' &= (\phi:n):H + U:n = (:H)\phi:n + U:n \\ &= (+) \frac{(:H)\phi}{0} :n \end{aligned}$$

Now, define the updating function u by

$$u:n = ( (+) \frac{(:H)\phi}{0} :n, H ) = (:H) (+) \frac{(:H)\phi}{0} :n$$

Then,  $\phi':n = Md:(\phi:n, u:n) = Md_{\frac{\phi}{0}}\phi:n$ . Therefore, the solution to

our problem, the new payroll file, is

$$\Phi' = Md \frac{\Phi}{u} \mid$$

where  $u = (,H)(+) \frac{(:H)\Phi}{u}$

CHECK ISSUING EXAMPLE: Suppose we wish to take the payroll file from the previous example and generate checks for the employees. We will assume that a function C is available such that  $C:(nm,p)$  returns a check in the amount p made out to the name nm.

SOLUTION: We will ignore overtime computations. Hence, if n is an employee number then  $\Phi:n:N$  is his name and

$$p:n = \Phi:n:H \times \Phi:n:R$$

is his pay. Hence, his check c:n is  $c:n = C:(nm,p:n) = C: \left( \frac{nm}{p:n} \right)$

$$= C: \left( \frac{(:N)\Phi:n}{p:n} \right) = C: \frac{(:N)\Phi}{p} \mid :n$$

Combining these we have the file F mapping employee numbers into checks:

$$F = C \frac{(:N)\Phi}{\frac{X:H}{X:R} \Phi} \mid$$

from which we can factor out the old payroll file:

$$F = C \frac{:N}{\frac{X:H}{X:R}} \mid \Phi$$

If we just want a set of checks, this is  $\vec{Lm}:F$ .

#### 14. Implementation Notes

The primary goal of our investigation has been to determine if relational programming is significantly better than conventional methods. It would be premature to devote much effort to implementation studies before it is even determined if relational programming is an effective programming methodology. However, a brief discussion of implementation possibilities is probably not out of line.

The most obvious representation of a relation is the extensional representation, in which all the elements of a relation or class are explicitly represented in memory. There are many kinds of extensional representations, such as hash tables, binary trees and simple sorted tables. Of course, performance can be improved through the use of associative memories and active memories (in which each memory cell has a limited processing capability).

Some relations and classes will be so large that it is uneconomical to represent them explicitly in memory. In these cases an intensional representation should be used. Here a class or relation is represented by a formula or expression for computing that relation or class. Operations on the class or relation are implemented as formal operations on the expression. This is feasible because of the simple algebraic properties satisfied by relations. It can be seen that an intensional representation is really just a variant of a lazy evaluation mechanism. Sometimes

an intensional representation is necessary; for instance, relations of infinite cardinality, such as the numerical operators and relations, require an intensional representation.

Although the programmer could be allowed to choose between extensional and intensional representations for his relations, this is not necessary. It is probably feasible, and certainly higher level, to have the system choose representations on the basis of cardinality estimates of the classes and relations involved. The algebra of relations is regular enough that many of these decisions can be made at compile time. Any that can't can be deferred to run-time when exact cardinality information is available.

## 15. Conclusions

Of course, we are not the first to propose introducing aspects of a relational calculus into programming. Space limitations prohibit a comparison with previous work, such as that by Feldman and Rovner [5], Codd [3] and Childs [2]. What does distinguish this investigation is the exclusive use of relations in a general purpose programming language. It is hoped that the preceding discussion has made plausible some of the advantages claimed for relational programming in the Introduction. Considerable work remains to be done in evaluating the effectiveness of a relational calculus as a programming tool. For instance, the

optimum set of combinators and relational operators must be selected. Another non-trivial problem is the selection of a good notation for the relational calculus. More from convenience than conviction we have used the notation of [8] and [1]. Making relational programming an effective tool will require designing a notation that combines readability with the manipulative advantages of a two-dimensional algebraic notation. This is all preliminary to any serious considerations of software or hardware implementation techniques. The reader who is interested in more details about programming in a relational calculus should consult [7].

#### 16. References

- [1] Carnap, R. Introduction to Symbolic Logic and its Applications, Dover, 1958.
- [2] Childs, D.L. Feasibility of a set-theoretic data structure based on a reconstituted definition of relation. IFIP 68 Proceedings, 420-430, North-Holland, 1969.
- [3] Codd, E.F. A relational model for large shared data banks, CACM 13, 6 (June 1970), 377-387.
- [4] Curry, H.B., Feys, R. and Craig, W. Combinatory Logic, I, North-Holland, Amsterdam, 1958.



- [5] Feldman, J.A. and Rovner, P.D. An Algol-based associative language, CACM 12, 8 (August 1969), 439-449.
- [6] MacLennan, B.J. Fen - an axiomatic basis for program semantics, CACM 16, 8 (August 1973), 468-474.
- [7] MacLennan, B.J. Programming with a Relational Calculus, Computer Science Department Technical Report, Naval Post-graduate School, 1981.
- [8] Whitehead, A.N. and Russell, B. Principia Mathematica to \*56, Cambridge, 1970.

# INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52M1 Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12